Sentiment Classification Using Complex Neural Networks

Carlos Velasco, ITC A01708634, Tecnológico de Monterrey Campus Querétaro

Abstract.- This paper explores the application of complex neural networks for sentiment classification of IMDb movie reviews, aiming to distinguish between positive and negative sentiments. By implementing and analyzing advanced deep learning architectures, I evaluate the effectiveness of various models and their ability to capture nuanced language patterns.

I. Introduction

The Internet Movie Database (IMDb) is a widely-used online resource for information on films, television shows, and actors. A popular feature on IMDb is its user review system, where millions of users rate and leave feedback on movies and shows. Each review is accompanied by a star rating, reflecting the reviewer's sentiment, and provides insights into the public's perception of each film. This extensive collection of reviews presents a valuable opportunity for sentiment analysis, particularly in identifying patterns and trends in public opinion.

For this project, I utilize the "IMDB Dataset of 50K Movie Reviews," a well-known benchmark dataset for sentiment classification tasks in natural language processing. This dataset comprises 50,000 IMDb movie reviews, evenly split into 25,000 for training and 25,000 for testing. Each review is labeled as either positive or negative, enabling a binary classification approach to sentiment analysis. Compared to earlier sentiment datasets, this dataset offers a substantial volume of data, making it ideal for testing complex models that require robust training data to capture subtle language cues associated with sentiment.

The objective of this study is to leverage complex neural networks to classify reviews as positive or negative with high accuracy. By applying deep learning techniques, I aim to explore the potential of neural networks to understand and process natural language effectively.

II. Data Preprocessing

Before feeding text data into a neural network, it requires thorough preprocessing to ensure consistent and meaningful input. This section details the key stages involved in preparing the IMDb reviews dataset for sentiment classification, covering data loading, standardization, tokenization, and vectorization.

1. **Download Dataset:** We start by importing the IMDb dataset, a collection of 50,000 labeled movie reviews split between positive and negative sentiments. After downloading the dataset, we load it using pandas to explore its structure. The dataset is then ready for cleaning and transformation.

```python
movie_reviews = pd.read_csv('a1_IMDB_Dataset.csv')
print(movie_reviews.shape)
movie_reviews.head()
```

2. **Text Standardization:** Raw text data often contains HTML tags, punctuation, and variations in letter case, which can introduce noise. To standardize the text, we first define a function remove_tags that replaces HTML tags with whitespace using

regular expressions. Additionally, we convert all text to lowercase to ensure uniformity, remove non-alphabetical characters, and reduce extra whitespace.

```
Python
tag = re.compile(r'<[^>]+>')

def remove_tags(text):
    return tag.sub('', text)
```

3. **Cleaning and Stopword Removal:** After standardizing, further cleaning is necessary to remove irrelevant elements. This involves removing non-alphabetical characters, filtering out single characters, and removing stopwords, which are common words (e.g., "and," "the") that do not contribute to sentiment analysis. The preprocess_text function carries out these tasks using NLTK's stopword list for English.

```
Python
import nltk
nltk.download('stopwords')

def preprocess_text(t):
    text = t.lower()
    text = remove_tags(text)
    text = re.sub('[^a-zA-Z]',
' ', text)
    text =
re.sub(r"\s+[a-zA-Z]\s+", ' ',
text)
    text = re.sub(r"\s+", ' ',
text)
    pattern = re.compile(r'\b('
+
r'|'.join(stopwords.words('engl
ish')) + r')\b\s*')
    text = pattern.sub('',
text)
    return text
```

4. **Applying Preprocessing to the Dataset:** Once the preprocess_text function is defined, we apply it to each review in the dataset. This cleans the text, preparing it for tokenization and vectorization in the subsequent steps.

```
Python
X = []
for i in
range(movie_reviews.shape[0]):

X.append(preprocess_text(movie_
reviews.iloc[i][0]))
```

Through these preprocessing steps, the text data becomes more structured, enabling the neural network to analyze the core sentiment content more effectively. The cleaned dataset is now ready for the final stages of tokenization and vectorization, which convert the text into a numerical format suitable for model training.

III. Train & Test Split

To evaluate the performance of our sentiment classification model, it's essential to split the dataset into training and testing subsets. This allows us to train the model on one portion of the data and assess its generalization ability on unseen data, helping to prevent overfitting and ensuring robust performance. In this project, we use an 80/20 split, where 80% of the data is used for training and 20% for testing.

We define the target variable, **y**, by converting the sentiment labels to binary values: 1 for positive and 0 for negative reviews. This binary representation simplifies the classification task, as the model only needs to

distinguish between two classes. The binary transformation is implemented as follows:

```python
y = np.array(list(map(lambda x:
1 if x=="positive" else 0,
movie_reviews['sentiment'])))
```

The resulting training and testing sets, X_train, X_test, y_train, and y_test, allow us to evaluate the model's accuracy in predicting positive or negative sentiment on new, unseen data.

```python
X_train, X_test, y_train,
y_test = train_test_split(X, y,
test_size=0.20,
random_state=42)
```

IV. Tokenization

Tokenization is a fundamental preprocessing step in natural language processing (NLP), especially for text classification tasks. It involves breaking down text into smaller units called tokens, often individual words, that can be processed by a machine learning model. In the context of neural networks, tokenization is essential because models work with numerical data, not text. Converting text into sequences of numbers enables the model to recognize and learn patterns within the data.

In this project, we use Keras's Tokenizer to transform words in the reviews into numerical sequences. The Tokenizer builds a vocabulary from the training data and then maps each word in the dataset to a unique integer. This process captures the unique vocabulary of the dataset, making it possible to convert each review into a sequence of word indices.

```python
word_tokenizer = Tokenizer()
word_tokenizer.fit_on_texts(X_t
rain)

X_train =
word_tokenizer.texts_to_sequenc
es(X_train)
X_test =
word_tokenizer.texts_to_sequenc
es(X_test)
```

The vocab_size refers to the total number of unique words (tokens) in the dataset, plus one to account for the padding token. Setting a vocabulary size allows the model to limit the words it considers, focusing only on the most common words to reduce computation and memory usage. This is especially helpful for large datasets with extensive vocabularies. In this project, vocab_size is calculated as:

Additionally, we know that reviews vary in length, so setting a consistent maximum length (max_len) ensures all sequences fed into the model are of uniform size. This prevents issues during model training and allows the model to focus only on a fixed number of words. In this project, we set max_len to 100, meaning only the first 100 words of each review are retained. Reviews with fewer than 100 words are padded, while longer reviews are truncated.

```python
vocab_size =
len(word_tokenizer.word_index)
+ 1

vocab_size

max_len = 100
```

To ensure uniformity, shorter reviews are padded with zeros to reach max_len. This padding is applied after the sequence

(padding='post'), adding zeros at the end of each sequence to standardize its length. Padding is crucial as it allows the model to process each review as a fixed-length sequence, regardless of its original length.

```
Python
X_train =
pad_sequences(X_train,
padding='post', maxlen=max_len)
X_test = pad_sequences(X_test,
padding='post', maxlen=max_len)
```

With tokenization and padding complete, the dataset is almost ready for input into the neural network. The tokenized sequences need to be converted into dense, meaningful representations through text embedding, which transforms individual words into vectors that capture semantic relationships. This step enhances the model's ability to understand the sentiment in each review based on the contextual similarity between words.

V. Text Embedding

Text embedding is a technique used to convert words into dense, continuous vectors that represent semantic relationships among words. Unlike traditional bag-of-words models, which treat words as discrete entities, embeddings capture subtle meanings, allowing models to understand similarities and differences between words. This process is crucial for natural language processing tasks, as it provides a way for neural networks to interpret and learn from text more effectively by leveraging the relationships between words.

In this project, we use the GloVe (Global Vectors for Word Representation) embedding method. GloVe is a widely-used pretrained embedding model developed by Stanford, where words are represented as vectors based on their co-occurrence statistics in a large corpus. The idea behind GloVe is that words appearing in similar contexts should have similar vector representations. This is especially beneficial for sentiment analysis, as words with positive and negative connotations will be represented by distinct vectors, helping the model distinguish between sentiments more accurately.

The GloVe embedding model is effective because it balances local context (how words relate within a specific sentence) with global context (how words relate across a broader corpus). This dual focus provides rich, generalized word representations that perform well across a variety of NLP tasks, including text classification.

To implement GloVe in our project, we use the pretrained glove.6B.100d embeddings, where each word is represented by a 100-dimensional vector. Here's how the embedding process is set up:

Loading the GloVe Embeddings

We start by loading the glove.6B.100d.txt file, which contains word embeddings for a large vocabulary. Each line in the file represents a word followed by its corresponding 100-dimensional vector. We load these embeddings into a dictionary, where each word is mapped to its vector.

```
Python
from numpy import asarray
from numpy import zeros

embeddings_dictionary = dict()
glove_file =
open('a2_glove.6B.100d.txt',
encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions =
asarray(records[1:],
dtype='float32')
```

```
    embeddings_dictionary[word]
= vector_dimensions
glove_file.close()
```

Creating the Embedding Matrix

Next, we create an embedding matrix that maps each word in our dataset's vocabulary to its corresponding GloVe vector. The matrix has dimensions (vocab_size, 100), where vocab_size is the number of unique tokens in our dataset, and 100 is the vector dimension. If a word in our vocabulary has a GloVe vector, it's placed in the embedding matrix. Words not found in GloVe are initialized as zero vectors.

```
Python
embedding_matrix =
zeros((vocab_size, 100))

for word, index in
word_tokenizer.word_index.items
():
    embedding_vector =
embeddings_dictionary.get(word)
    if embedding_vector is not
None:
        embedding_matrix[index]
= embedding_vector
```

With this embedding matrix, each tokenized sequence from our dataset is now mapped to meaningful, dense vectors. This embedding process enhances the neural network's ability to capture semantic relationships between words, ultimately improving the accuracy of sentiment classification by giving the model contextual understanding of the text. The dataset is now fully prepared for input into the neural network for training and evaluation.

VI. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of deep learning model originally designed for image processing but has proven effective in natural language processing tasks as well. CNNs use convolutional layers to automatically capture complex patterns in data by applying filters that detect local relationships. For sentiment analysis, CNNs are advantageous because they can identify key features in text, such as sentiment-bearing words or phrases, that might indicate positive or negative sentiment.

CNNs are well-suited for text classification because of their ability to learn patterns in sequences, even in contexts where words appear in different positions. This quality makes CNNs effective for identifying sentiment across various reviews, as they can capture subtle yet impactful phrases and their relationships to one another within the text.

In this project, I implemented a CNN for sentiment classification as follows:

Embedding Layer

The model begins with an embedding layer that uses the GloVe embedding matrix we created earlier. This layer converts each word index in the input sequences into a 100-dimensional vector based on the pretrained GloVe embeddings. Setting trainable=False keeps the embeddings fixed, leveraging GloVe's pretrained semantic relationships without modifying them during training.

```
Python
embedding_layer =
Embedding(vocab_size, 100,
weights=[embedding_matrix],
input_length=max_len,
trainable=False)
cnn_model.add(embedding_layer)
```

Convolution and Pooling Layers

Next, a 1D convolutional layer (Conv1D) with 128 filters and a kernel size of 5 is added. This layer moves across the text data and applies

filters that capture meaningful patterns and local relationships within the sequence. The activation function used here is ReLU, which adds non-linearity and helps the model learn complex representations. After the convolution layer, a GlobalMaxPooling1D layer reduces the dimensionality by selecting the maximum value from each filter, effectively summarizing the most important features detected.

```python
cnn_model.add(Conv1D(128, 5,
activation='relu'))
cnn_model.add(GlobalMaxPooling1
D())
```

Dense Layers

After pooling, a dense layer with 10 neurons and ReLU activation is included, adding another layer of abstraction to the feature representations learned in the previous layers. Finally, the output layer contains a single neuron with a sigmoid activation function, which is ideal for binary classification tasks like ours, as it outputs probabilities indicating whether a review is positive or negative.

```python
cnn_model.add(Dense(10,
activation='relu'))
cnn_model.add(Dense(1,
activation='sigmoid'))
```

Compiling and Training the Model

The model is compiled with the Adam optimizer and binary cross-entropy loss function, suitable for binary classification tasks. The model was trained over six epochs with a batch size of 128, using a validation split of 20% to assess its performance during training.

```python
cnn_model.compile(optimizer='ad
am',
loss='binary_crossentropy',
metrics=['acc'])
cnn_model_history =
cnn_model.fit(X_train, y_train,
batch_size=128, epochs=6,
verbose=1,
validation_split=0.2)
```

The CNN model is now trained on our dataset and ready for evaluation. In the next section, we will discuss the results and evaluate the model's performance on sentiment classification for IMDb reviews.

VII. Results of CNN

This section presents the performance metrics of our CNN model, covering accuracy, loss, confusion matrix, precision, recall, and F1-score. These metrics provide a quantitative understanding of the model's performance, laying the groundwork for further analysis in the following sections.

Accuracy

Accuracy is the ratio of correct predictions to the total number of predictions. During training, we observe that accuracy increases with each epoch, reaching approximately 97% on the training set by the final epoch, while the validation accuracy is around 84%. On the test set, the accuracy score is 84.46%. Accuracy serves as an initial indicator of how well the model is performing overall but does not provide specific insights into each class's performance.

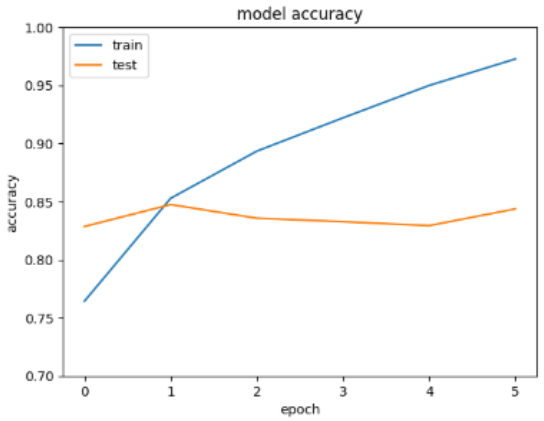

Figure 1. CNN model accuracy in train and test sets

Loss

Loss quantifies the error between predicted and actual outcomes. The model's loss decreases as it trains, indicating that it is learning and refining its predictions. For the test set, the final loss value is 0.392. This metric is essential in guiding the model's optimization process during training and in assessing generalization on unseen data.

Confusion Matrix

The confusion matrix breaks down the model's predictions into four categories:

True Negatives (TN): Correctly classified negative reviews (4150).
False Positives (FP): Negative reviews incorrectly classified as positive (811).
False Negatives (FN): Positive reviews incorrectly classified as negative (743).
True Positives (TP): Correctly classified positive reviews (4296).

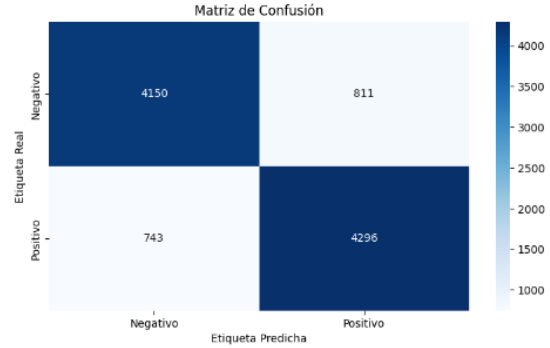

Figure 2. CNN model confusion matrix

Visualizing the confusion matrix highlights the model's strengths and weaknesses in handling each sentiment class.

Precision

Precision measures the accuracy of positive predictions, calculated as “Precision=TP / (TP + FP)”.

Here, the model achieves a precision of 0.841, meaning 84.1% of positive predictions are correct. Precision is particularly useful when false positives are costly or undesirable.

Recall

Recall (or sensitivity) assesses the model's ability to detect all positive samples, calculated as “Recall = TP / (TP + FN)”

The model’s recall is 0.853, indicating that it correctly identifies 85.3% of actual positive reviews. Recall is essential when we want to minimize false negatives.

F1-Score

The F1-score is the harmonic mean of precision and recall, offering a balance between the two.

With an F1-score of 0.847, the model shows a balanced performance in terms of both precision and recall.

These metrics outline the model's performance and offer insight into its ability to handle the sentiment classification task. In the next section, we will interpret these results and explore areas for improvement and further analysis.

VIII. Result’s Interpretation

Model Fit and Overfitting

The model achieves high accuracy on the training set (97%) but a notably lower accuracy on the test set (84%). This difference suggests overfitting, where the model has learned the training data too well, potentially memorizing specific patterns or noise.

Overfitting occurs when a model's complexity allows it to capture details unique to the training set that do not generalize well to new, unseen data. Here, the model's ability to generalize is limited, as indicated by its reduced performance on the test data. In practical terms, overfitting implies that our CNN model struggles to accurately predict sentiment on reviews it has not seen before, limiting its usefulness in real-world applications.

Model Bias

Looking at the confusion matrix, we observe a relatively balanced distribution between true positives (TP) and true negatives (TN), which indicates that the model does not exhibit significant bias towards one class over the other. The 1:1 ratio of positive to negative reviews in the original dataset contributes to this balance, as it prevents the model from developing a preference for one sentiment class. When models are trained on well-balanced datasets, they are less likely to be biased towards the majority class, improving their performance in handling each category fairly. In this case, the model treats positive and negative sentiments with similar accuracy, reflecting the balanced nature of the data.

Model Variance

Variance refers to the model's sensitivity to small changes in the training data. A high-variance model can fit its training data accurately but may struggle with new data, leading to inconsistent performance. Here, our CNN model displays high variance, as evidenced by its strong performance on the training set but weaker results on the test set. The model's significant reduction in accuracy and increased test loss demonstrate its difficulty in adapting to unseen data, likely because it is tightly fitted to the specific patterns of the training samples. High variance usually stems from overfitting and is common in complex models, like neural networks, which can learn detailed data patterns that do not generalize well.

IX. CNN Result's Conclusion

In summary, the CNN model's performance reflects a classic case of overfitting, where high variance limits its ability to generalize beyond the training data, even though it does not exhibit bias towards one sentiment class. Moving forward, addressing overfitting—through techniques such as regularization, dropout layers, or using a simpler model architecture—could improve the model's generalizability and reduce its variance.

X. Recurrent Neural Network (LSTM)

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data by maintaining a hidden state that captures information about previous inputs. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing them to retain memory of previous inputs and thus model temporal dependencies in data. This characteristic makes RNNs particularly well-suited for tasks involving sequences, such as language modeling, speech recognition, and sentiment analysis.

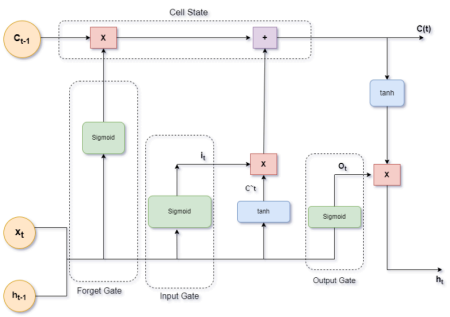

Figure 3. Typical LSTM Architecture

Long Short-Term Memory (LSTM) networks are an advanced variant of RNNs that address the vanishing gradient problem, which hampers the learning of long-term

dependencies in standard RNNs. LSTMs introduce specialized gating mechanisms—namely, the input gate, forget gate, and output gate—that regulate the flow of information, enabling the network to retain or discard information as needed. This architecture allows LSTMs to capture intricate patterns and dependencies over extended sequences, making them highly effective for natural language processing tasks where context and order are crucial.

In the context of text embedding and sentiment classification, LSTMs offer significant advantages. Text data is inherently sequential, with the meaning of a sentence often depending on the order and relationship of words. LSTMs excel at modeling these sequential dependencies, allowing the network to understand the context and nuances of the text more effectively than models that treat input data as independent tokens. By leveraging text embeddings, such as GloVe vectors, LSTMs can process dense vector representations of words, capturing both semantic and syntactic information to improve classification performance.

In this project, I implemented an LSTM-based model for sentiment classification as follows:

Embedding Layer
Similar to the CNN model, the LSTM model begins with an embedding layer initialized with the pretrained GloVe embeddings. This layer transforms each word index in the input sequences into a 100-dimensional vector, providing a rich semantic representation of the text. Setting trainable=False ensures that the pretrained embeddings remain unchanged during training, preserving their semantic integrity.

```
Python
embedding_layer =
Embedding(vocab_size, 100,
weights=[embedding_matrix],
input_length=max_len,
trainable=False)
lstm_model.add(embedding_layer)
```

LSTM Layer
The core of the model is the LSTM layer, which consists of 128 neurons. This layer processes the sequential data, capturing temporal dependencies and contextual information from the embedded word vectors. The LSTM layer's ability to retain information over long sequences allows the model to understand complex sentiment patterns within the reviews.

```
Python
lstm_model.add(LSTM(128))
```

Output Layer
Following the LSTM layer, a dense output layer with a sigmoid activation function is added. This single neuron outputs a probability score between 0 and 1, indicating the likelihood of a review being positive. The sigmoid activation is ideal for binary classification tasks, providing a clear decision boundary for sentiment prediction.

```
Python
lstm_model.add(Dense(1,
activation='sigmoid'))
```

Compiling and Training the Model
The model is compiled using the Adam optimizer and binary cross-entropy loss function, which are well-suited for binary classification problems. Training is conducted over six epochs with a batch size of 128, utilizing a validation split of 20% to monitor the model's performance on unseen data during training.

```python
# Compile the model
lstm_model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['acc'])
lstm_model.summary()

# Train the model
lstm_model_history =
lstm_model.fit(X_train,
y_train, batch_size=128,
epochs=6, verbose=1,
validation_split=0.2)
```

The LSTM model leverages its ability to understand and retain sequential information, enhancing its capacity to perform accurate sentiment classification based on the contextual relationships within the movie reviews. In the next section, we will present the results of this LSTM model, comparing its performance to the previously discussed CNN model.

XI. LSTM Results

The LSTM model was trained for six epochs, with the following results:

Training Accuracy and Loss

Over six epochs, the training accuracy increased from 67.45% to 87.32%, while the training loss decreased from 0.5821 to 0.3015, indicating effective learning during training.

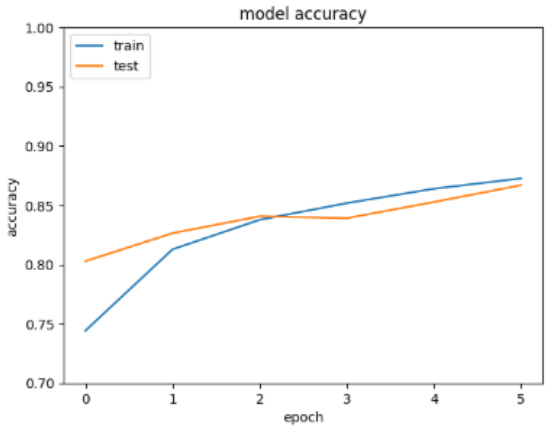

Figure 4. LSTM accuracy in train and test sets.

Validation Accuracy and Loss

The validation accuracy started at 80.29% and progressively improved to 86.70% by the sixth epoch. Similarly, the validation loss declined from 0.4472 to 0.3158, showing consistent improvement on the validation set.

Test Set Performance

The model achieved a test accuracy of 86.14% with a test loss of 0.3205, reflecting its ability to generalize to new, unseen data.

Confusion Matrix and Metrics

True Negatives (TN): 4,073
False Positives (FP): 888
False Negatives (FN): 498
True Positives (TP): 4,541

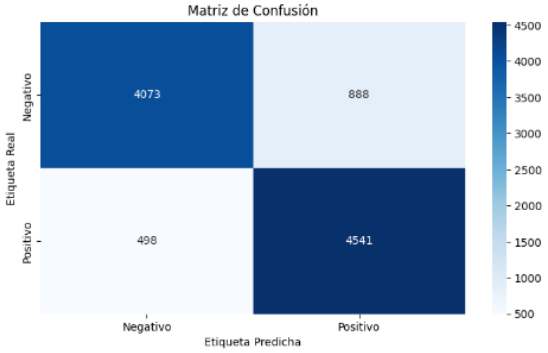

Figure 5. LSTM model confusion matrix

Precision: 0.836
Recall: 0.901
F1-Score: 0.868

XII. LSTM Results' Interpretation

Fitting

The LSTM model demonstrates a good fit to the data, balancing both training and validation accuracies without severe overfitting or underfitting. While training accuracy is slightly higher than validation accuracy (87.32% vs. 86.70%), the difference is minimal, indicating that the model does not simply memorize training data but rather learns generalizable patterns. This is further

validated by the test accuracy of 86.14%, which is close to the validation accuracy.

Bias

There is minimal bias in this model, as evidenced by the relatively high recall (0.901) and balanced true positive (TP) and true negative (TN) counts in the confusion matrix. The dataset's balanced nature between positive and negative classes supports this low bias, as the model can accurately classify both types without strong preference for one class over the other.

Variance

The model displays low variance due to its consistent performance across training, validation, and test sets. The close alignment of these accuracy metrics suggests that the model is not highly sensitive to fluctuations in the training data and generalizes well to new data. The F1-Score of 0.868 further highlights this stability, as it combines both precision and recall, showing that the model is consistently accurate in both predicting positive cases and correctly identifying true positives.

XIII. LSTM Results' Conclusion

In conclusion, the LSTM model performed well in capturing patterns in the text data, achieving high accuracy and balanced performance metrics across training, validation, and test sets. The model's strong recall and F1-Score indicate its effectiveness in identifying positive cases while maintaining a balanced classification approach, likely aided by the original dataset's 1:1 ratio. Minimal overfitting and low variance further validate the model's robustness, suggesting it has successfully learned generalizable features from the data without being overly influenced by specific training instances. Overall, the LSTM model's results reflect a reliable approach to sentiment classification in text-based data, highlighting its potential for further applications and improvements.

XIV. References

Gonzalez, L. (2022, 7 septiembre). Sesgo y Varianza en Machine Learning. Aprende IA. https://aprendeia.com/bias-y-varianza-en-machine-learning/

Kim, J. (2018, 12 junio). Understanding how Convolutional Neural Network (CNN) perform text classification with word embeddings. Medium. https://towardsdatascience.com/understanding-how-convolutional-neural-network-cnn-perform-text-classification-with-word-d2ee64b9dd0b

Skillcate AI. (2022, 27 julio). Sentiment Analysis with LSTM | Deep Learning with Keras | Neural Networks | Project#8 [Vídeo]. YouTube. https://www.youtube.com/watch?v=oWo9SNcyxII

Thomas, E. B. (2023, 26 julio). Understanding LSTM: An In-depth Look at its Architecture, Functioning, and Pros & Cons. https://www.linkedin.com/pulse/understanding-lstm-in-depth-look-its-architecture-pros-babu-thomas/